This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCAD.2019.2907912

Efficient Job Offloading in Heterogeneous Systems through Hardware-assisted Packet-based Dispatching and User-level Runtime Infrastructure

Othon Tomoutzoglou*, Dimitris Mbakoyiannis*, George Kornaros*, and Marcello Coppola†
*Inform. Eng. Dept., Technological Educational Institute of Crete, Iraklio, GR †STMicroelectronics, Grenoble, FR

***Abstract*—Emerging heterogeneous systems architectures increasingly integrate general-purpose processors, GPUs, and other specialized computational units to provide both power and performance benefits. While the motivations for developing systems with accelerators are clear, it is important to design efficient dispatching mechanisms in terms of performance and energy while leveraging programmability and orchestration of the diverse computational components. In this article we present an infrastructure composed of a hardware, General, Packet-based Processing-dispatching Unit, named GPPU, and of an associated runtime that facilitates user-level access to GPPU objects such as packets, queues and contexts. Hence, we remove drawbacks of traditional costly user-to-kernel-level operations, low-level accelerator subtleties that hinder programming productivity, along with architectural obstacles such as handling accelerators' unified virtual address space. We present the design and evaluation of our framework by integrating the GPPU infrastructure with data streaming type accelerators, image filtering and matrix multiplication, tightly coupled to ARMv8 architecture via unified virtual memory. Under scaling workload our proposed dispatching methods can deliver 3.7× performance improvement over baseline offloading, and up to 4.7× better energy efficiency.**

***Index Terms*—Packet-based dispatching, CPU-Accelerators unified address space, Hardware-assisted offloading, User-level embedded systems dispatching**

I. Introduction

In the last decade together with the explosion of integrating multiple cores in a single chip, an architectural trend is the growing prominence of heterogeneous architectures. As scaling the number of processors does not always translate to linear speedup, heterogeneous systems aim to unlock the performance and power efficiency of the parallel hardware resources including CPUs, GPUs, DSPs, FPGAs, fabrics and fixed function accelerators in today's complex Systems-on-Chip (SoCs). However, even though physically putting together CPUs, GPUs and other accelerators on the same chip or platform, the available programming models still consider them separated.

Different techniques emerge to amortize heterogeneity in terms of architectural differences and programmability. Multiple processing units with their own memory may incur potentially penalty to access the data if these are not in close proximity, especially if they reside in a different address space. To tackle this challenge *fused* architectures, that is multi-core CPUs and many-core GPUs are integrated on a single chip with a shared on-chip L3 cache, to produce what AMD calls accelerated processing units (APUs) [1], providing the opportunity to leverage the high computational power of the GPUs. In addition, inspired by these architectures Heterogeneous System Architecture (HSA) Foundation [2] proposes a unified virtual address space and coherent shared memory spanning the APUs, enabling user-level task queues to reduce offloading latency.

Applications in heterogeneous architectures take advantage of hardware accelerators and GPUs, by offloading computations, intensive portions of their execution to hardware accelerators, either fixed-function or programmable such as a GPU. These offloaded computation tasks are referred as *jobs* in this article. A job consists of two parts, the *kernel*, that is, the executable code for a programmable accelerator, and the corresponding data. When the CPU dispatches a task to the hardware accelerator or to the GPU, it is usually necessary to pass through an OS service and an OS kernel driver before finally reaching the final target, which causes non-negligible performance degradation. As accelerators commonly execute outside the processor, they usually lack access to processor's virtual address space and invoking the accelerator may require pinning data in memory and translating virtual addresses in advance of launching the accelerated computation. Researchers have proposed single CPU thread to assist in prefetching data for many GPU threads [3] or channels, i.e., multi-producer/multi-consumer data queues that aggregate fine-grain work items before scheduling to the GPUs [4]. Furthermore, to make effective use of multi-core CPUs and GPUs, toolkits offer a different approach to parallelization [5], exploiting for instance an intuitive fork-join model and enabling locality friendly coarse-grained parallelism (OpenMP), or expressing fine-grained parallelism (OpenCL), enabling for example, implicit or explicit vectorization at the compiler level.

In this work we address the optimization of heterogeneous computing in terms of performance and energy consumption. *Heterogeneous* refers to platforms that use more than one kind of processors and/or specialized fixed-function or programmable accelerators. We built an innovative infrastructure to offload, i.e., dispatch, computational intensive jobs from the host processor to the heterogeneous computation nodes, which infrastructure comprises a hardware-assisted Generic Packet Processing Unit (GPPU) and a corresponding *runtime*, hereafter called as Architected Queuing Language-aware System Manager (*AQLSM*). The GPPU allows a programmer to write

Manuscript received September 1, 2018; revised December 31, 2018. Corresponding author: G. Kornaros (email: kornaros@ie.teicrete.gr).

Copyright (c) 2019 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCAD.2019.2907912

applications that seamlessly integrate different computing units removing today's hurdles such as separate memory spaces between CPU and non-virtualized hardware accelerators. In particular, the contributions of this work are summarized next. The GPPU hardware and runtime software components:

- offer user-level offloading to computation nodes in a system without costly OS system calls
- allow sharing of virtual address space between all processing components in the system (compatible to the HSA initiative) in order to remove the need of explicit copies; enable efficient data sharing between host CPU and hardware accelerators while removing the need of a complex Input/Output Memory Management Unit (IOMMU)
- accelerate the dispatching process with dedicated hardware-assisted unit which hides cumbersome configuration and monitoring of diverse attached accelerators
- remove the programming complexity while featuring a unifying way to optimize data and code transfers across different heterogeneous processing units and thus making applications to be easily re-targeted to different platforms

Our GPPU infrastructure assumes applications that are launched on the host CPU (or cluster of CPUs) and intermittently offload jobs to an accelerator. Sharing opportunities arise due to two reasons. First, a job might have completed an offload and is running on the host CPU leaving the accelerator free. Second, a job offload may not be using all of the cores on the accelerator, or a number of accelerators which are attached as sub-nodes to a GPPU, therefore allowing another job to potentially use the free accelerator cores. The focus of this work is not the optimal partitioning and mapping of an application to the multiple heterogeneous computing units. In addition, the programmer or the OS orchestrator must evaluate when an offload computation will outperform one that is local by forecasting the local cost (execution time and energy consumption for computing locally) and remote cost for computing remotely and transmission time for the input/output of the computation to/from the remote accelerator.

The rest of this paper is organized as follows. In section II, we introduce previous works on improving programmability and offloading costs in Heterogeneous SoCs. In section III, we give an overview of our work. Section IV presents the design and Implementation of the GPPU infrastructure. In section V we evaluate our proposals in terms of performance and energy consumption and finally, we conclude our work in Section VI.

II. Related Work

Optimizing communication and automatic data management have been addressed in the context of CPU-GPU program co-execution, first, through using compiler-assisted techniques to transfer data to the appropriate memory space [6] [7]. In the StarPU runtime [8], applications submit computational tasks, forming a task graph, with CPU and/or GPU implementations, and StarPU schedules these tasks and associated data transfers on available CPUs and GPUs. StarPU optimizes data transfers using prefetching and overlapping. In the same scope semi-automatic techniques, such as Global Memory for Accelerators (GMAC) require programmers to add annotations [9]. By sharing the virtual address space of CPU and accelerators, researchers have proposed a user-level library, GMAC, to make heterogeneous systems easier to program while reducing performance penalties [10]. It is not though guaranteed to successfully map the accelerator's memory to the same range of virtual memory address space. Unified Virtual Addressing (UVA) [11] abstracts away from the programmer the actual location of data, whether on any of the GPUs or on the CPU. On top, we propose methods to eliminate time-consuming operations such as user- to kernel- space data transfers, kernel space synchronization, queue management and dispatching.

Industrial Accelerators Coupling. Fixed-function accelerators commonly use scratchpad memories and Direct Memory Access (DMA) engines or caches. Cache-based memory hierarchy is inefficient for bulk data movement in I/O bound applications, but can be more efficient at on-demand, fine-grained data transfer for applications, and depending on the applications memory access patterns [12]. In industry, IBM introduced the Coherent Accelerator Processor Interface (CAPI) on POWER8 systems to provide a high-performance solution for implementing client-specific computation-heavy algorithms on FPGAs [13]. Additionally, the Multi-Accelerator Platform Engine for Baseband (MAPLE-B) consists of a programmable-system-interface which is a programmable controller with DMA capabilities and signal-processing accelerators attached using an internal interface [14] [15]. Intel proposes the QuickAssist Technology Accelerator Abstraction Layer to simplify the use and deployment of tightly-coupled accelerators [16] [17] via the Front Side Bus, but mainly for offloading computationally intensive compression and encryption tasks. By using a reduced functionality or "bare-bone" Application Programming Interface (API), named *dataplane API*, or by batching requests, Intel aims to reduce offload latency.

Data Transfers Optimizations. To tackle offloading overheads, researchers have also proposed dynamically aggregating asynchronously produced fine-grain work into coarser-grain tasks leveraging the GPU's control processor to manage those queues [4]. Optionally, engineers develop hardware-assisted DMA and I/O read and write access methods along with on-chip microcontrollers inside the GPU to offer effective solutions in terms of reducing the data transfer latency for concurrent data streams [18]; Fujii et al., [18] showed that direct I/O operations are faster than using DMA controllers for small data transfers. Hardware support for optimizing different ISA-based heterogeneous systems is also proposed in a variety of contexts, by accelerator management to mitigate memory latencies during data transfer [19], or by optimizing intra-node communication using DMA assistance [20]. Similarly, in [21] we proposed an open-source framework for optimized data transfers over PCIe-attached accelerators via exploiting scheduling and parallelized, pipelined DMA transfers, all controlled by hardware customized components at the FPGA fabric side. However, in the work in this article we introduce packet-based dispatching for tightly-coupled accelerators with a user-level runtime which seamlessly integrates different acceleration units.

Device Virtual Address Space Access. Modern systems are equipped with IOMMUs [22] [23], to support address translation for loosely-coupled devices, including customized

Copyright (c) 2019 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCAD.2019.2907912

accelerators and GPUs. To provide access to virtual address space some GPUs include highly complex IOMMUs; 4-ported private TLBs and improved page walk scheduling have been proposed [24], while authors have also introduced a highly threaded page walker to handle bursts of TLB misses [25]. Instead, we propose a novel mechanism that is totally free from an IOMMU component, by taking advantage of the processor's translation functionality also for the accelerators, and by using a simplified address translation mechanism. Earlier, we introduced the concept of packet-based hardware dispatching in [26], which integrates a runtime based on kernel-space interprocess communication facilities. A key-goal of this work is to support a fully distributed user-space runtime (via ARMv8 atomic primitives) and offer two dispatching alternatives to remove IOMMU overheads: (a) use a dedicated system memory partition for data and queue contexts (as in [26]) and, (b) utilize the full-system memory, while exploiting the CPU's MMU to deliver a unified virtual memory view to the accelerators as well.

III. Job Dispatching to Accelerators

To achieve efficient dispatching the proposed infrastructure consists of the Generic Packet Processing Unit (GPPU) and of the Architected Queuing Language-aware System Manager (AQLSM) runtime, which work synergistically to enable workload dispatching in a packet-based fashion and, allow the programmer to handle hardware accelerators in a unifying, transparent and low complexity way. The GPPU controls one or even multiple different accelerators (figure 1) and abstracts the diversity of the functionality of these custom accelerators by presenting a single general-purpose API to the host CPU and hiding the accelerators details.

The design of the GPPU as a hardware component addresses the following objectives.

- enable all computing units in a heterogeneous system to operate in their own virtual address space
- allow direct interfacing to user space applications without expensive system calls
- offload the CPU from scheduling of dispatching operations and monitoring of accelerators
- seamless dispatching, unaware of the type of the computing unit; i.e., the CPU can dispatch a job to the GPU, or the other way round

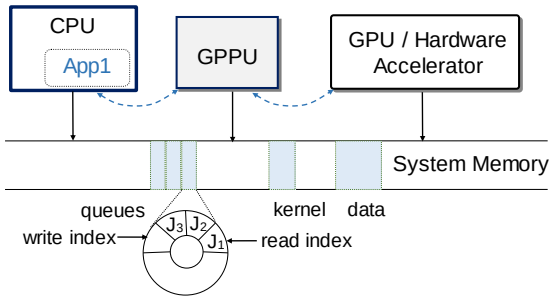

Fig. 1. Dispatching jobs (kernels and data) from CPU to accelerators and from accelerators to CPU through using the GPPU in a packet-based communication and user-level accessible circular queues

The supported programming model defines job offloading through command queue objects. These queues are allocated at runtime and they are accessible by applications running on the host CPU, at user level, in applications' virtual address space. Each queue contains packets, i.e., commands, as defined in specifications of HSA Foundation [2] (AQL packets). Both queues and packets are allocated and de-allocated by applications through the AQLSM runtime infrastructure. Queues are *semi-opaque objects* for which the AQLSM maintains a queue context, i.e., circular buffer related information. The visible part of a queue context includes queue type (i.e., allocated to a single application or shared by multiple applications), doorbell signal, queue size, read and write indexes and a queue index. The invisible part of a queue context contains sensitive and error-prone information, i.e., the queues and packets virtual and physical addresses, which are accessed only by the AQLSM (CPU), the GPPU and the accelerators. A set of queues are pre-assigned to a particular GPPU and the AQLSM allocates each queue to user requests. The GPPU on the other hand is responsible for the binding of a queue to an available accelerator from the set of accelerators which are attached to a GPPU; this binding is independent of the kind of accelerator. The command packets are 64-byte fixed-size data structures following the *kernel dispatch packet* format [2].

As shown in figure 1, the GPPU performs the following functions: (i) it manages requests from user-level applications to offload jobs to accelerator engines; the requests are performed in the form of command packets which are placed in circular queues until the GPPU serves these queues, (ii) it mediates the process to offload jobs (kernel and data) from user application memory space to a hardware accelerator and in the opposite direction, to monitor the accelerator process and notify the application of the outcomes of these jobs.

A user-level queue is a shared memory space between CPU and GPPU and implements a one-way communication, either from the CPU to the accelerator, or from the accelerator to the CPU, via the GPPU. Figure 2 depicts different options, which a user application may utilize to offload workload to the accelerators; the dark shaded block (ACC2) indicates an accelerator of different functionality compared to the light shaded block (ACC1). In this article we opted for option (c),

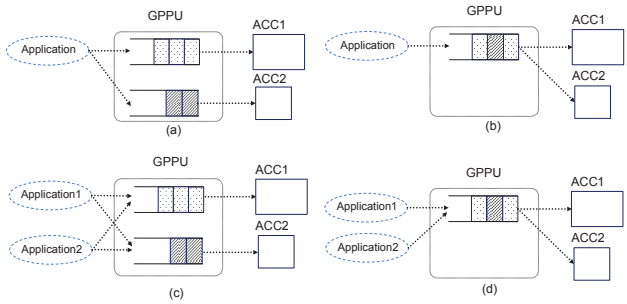

Fig. 2. Feasible offloading methods enabled by the GPPU

and we intend to evaluate queue sharing in future work. Both CPU and GPPU maintain an internal state to enable race-free accesses to read and write indexes of the command queue in a consistent way. A complete offload operation involves the *launch*, *active* and *completion* phases. The launch phase includes the operations to fill and successfully enqueue a packet; then, during the active phase, the GPPU selects an

Copyright (c) 2019 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCAD.2019.2907912

eligible packet and processes its contents to submit the job to the appropriate accelerator. Finally, in the completion phase the GPPU notifies the application of the acceleration outcome.

The CPU initializes a queue context in a local memory (register file) inside the GPPU to configure the communication protocol parameters (*writeIndex*, *readIndex*, *queueSize*). Then, a user application is allowed to enqueue command packets to the ring buffer queue using the unique queue index and packet index. A new packet index is obtained by calling the GPPU runtime, i.e., the *AQLSM*. The application uses this queue unique index to find the available packet within the ring buffer. This is done through the AQLSM API. Internally, the AQLSM creates the appropriate virtual address, called *userVA*. The application, populates the packet including parameters and pointers to the workset that needs to offload. The workset involves the kernel code and the data to process when a programmable accelerator is used, or only the data when a fixed-function hardware accelerator is connected. Finally, the application creates a logical signal, called *doorbell* to notify the GPPU that the packet is ready to be processed. A signal can be actually considered a shared memory location that contains an integer, which when modified triggers an event in the GPPU. The GPPU will dispatch all packets from a circular queue until a barrier packet is identified. It is not necessary for all the packets which have been dispatched to reach their completion phase before more packets from the same queue can be launched.

When the doorbell has been received, the GPPU retrieves the packet from the head of the queue, which is located at the readIndex. At this point the packet is processed by the GPPU on the basis of the packet type. We support two types of packets: the dispatch which describes a new job, and the barrier, which includes a command for the GPPU. In case of a dispatch packet, the pointers in the packet are extracted to enable the wake up of the accelerator. When the accelerator has completed processing, the GPPU can update the readIndex, and can set the packet as invalid. Finally, the signal included in the packet will be reset by the GPPU, so the application in the host can be notified to use the outcome data produced by the accelerator. Of course the application can asynchronously submit multiple jobs to the same queue, either one-by-one or in a batch and continue its operation . The GPPU is associated with multiple AQL queues, independently of the number of accelerators.

A. System Memory Management Using a GPPU

Applications running on the host CPU can offload computation kernels onto a customized compute engine or GPU to get accelerated. Depending on the framework used (i.e., OpenCL/GL, CUDA) the offload process requires memory allocation, explicit kernel and data copying from the CPU to the accelerator address space, kernel launch, data copying from the accelerator to the CPU, and freeing of the allocated memory space. In modern HSA-style architecture the convenience of a unified virtual address space removes the need for maintaining two copies of the same data in both host and device address spaces. By simply passing a pointer to the shared data in the virtual address space to and from the accelerator essentially eliminates unnecessary and costly copies. In [27] we can see the benefits of using unifying address space in heterogeneous SoCs. However, since accelerators actually need to access the data in their physical memory space a virtual-to-physical address translation is required. Modern SoCs have introduced hardware IOMMUs to enable devices to access the virtual memory that the host CPU supports [28] [29]. Even though recent advances in IOMMU designs enable translation caching to compensate for the long latencies of page table walking, the silicon cost of the IOMMU along with its performance overhead creates a significant challenge for loosely-coupled accelerator architectures. In the scope of accessing CPU from accelerators, recently Vesely et al., [30] proposed to designate a portion of memory as the syscall area for GPU to store system call arguments and information and then GPU to interrupt the CPU (by relying on the ability of the GPU to interrupt the CPU) and send the ID number of the wavefront issuing the system call. We envision a more structured way for this process using packet-based communication to the CPU and directly notify user-space application in contrast to OS service as in [30]. In this work we have developed both interrupt-driven and CPU-polling service to identify acceleration completion.

In addition, the proposed GPPU framework requires also to expose the physical address space of queues and corresponding context to the virtual memory of the applications. Hence, the unified virtual address space between host CPU, GPPU and accelerators requires an efficient mechanism for virtual to physical address translation. In this section we present two solutions developed to tackle memory management issues in this scope. Both techniques are the only ones known to the authors that totally eliminate the complexity and incurred overheads of using IOMMUs. In contrast to implementing a dedicated IOMMU for accelerators, where synchronization with the MMU of the host CPU is needed on page fault handling, our approach requires no additional support for system-level correctness issue.

1) System Memory Partitioning: One option to facilitate memory management in SoCs that integrate loosely-coupled accelerators is to reserve one partition of the physical memory for communicating data between user applications and accelerators. We call this approach as *SMP* method hereafter. The OS lives in the rest of the physical memory. Applications can access this reserved partition with the aid of the AQLSM and OS through the *mmap()* system call, which call is used only at the pre-offloading initialization phase. Even if the downside is the non-optimal utilization of the physical memory, nevertheless, devices can directly access this partition free from virtual-to-physical translations, while the applications can also access this memory through the AQLSM that does the mapping to applications virtual address space.

As figure 3 shows, a partition starting from *base* address till the high end of system memory is dedicated to store the queues and the applications data. This memory splitting occurs during the boot time. When an application requests a queue via *aqlsm_queue_create()* of the AQLSM API, then the AQLSM exports a virtual address pointer (userVA) to the queue data buffer by simply adding the appropriate offset to the partition base address.

Copyright (c) 2019 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCAD.2019.2907912

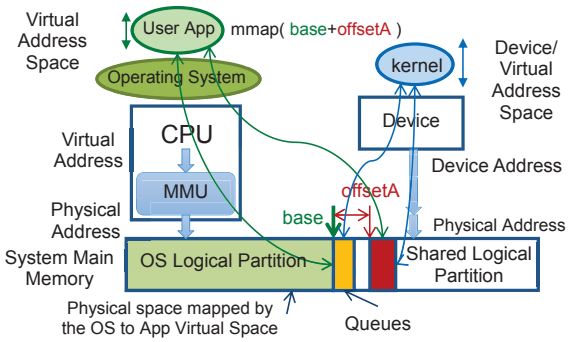

Fig. 3. Dispatching kernels to accelerators through the GPPU infrastructure in SMP fashion; the OS virtualizes the left shaded partition applications, while the AQLSM runtime utilizes the right (higher) memory partition for queues and application data for offloads

All queue data buffers are maintained in a logical partition that starts at the base address of the reserved partition, while next lies the logical partition for the application data. Figure 3 shows an example where an application acquires a queue object and will use a logical partition for the application data which corresponds with this particular queue. The AQLSM exports this data partition that is located at *offsetA* in the application virtual address space. Notice that by using mmap() call the virtual address returned corresponds to physically contiguous address space.

The GPPU on the other hand that works using physical addresses, accesses the same objects, i.e., *queues*, *kernels* and *data*, through using the physical addresses which are formed by adding the memory partition's base address and the corresponding offset. For instance, to access a particular packet, the GPPU uses the queue index (*queueIndex*) and the packet index (*packetIndex*), which are absolute numbers and calculates the physical address (PA) of the packet as follows.

$$PA_{packetIndex} = basePA + queueIndex * MaxQueueSize + packetIndex * PacketSize$$

where $MaxQueueSize$ represents the maximum queue size supported in the system. Our SMP approach is clearly straightforward and, since the data are physically contiguous, accessing them does not undergo any translation and is henceforth faster. Platform-wise the GPPU decapsulates a packet and delivers the physical address of the kernel to a programmable accelerator or of the data to a fixed-function hardware accelerator. This mediation function depends largely on the accelerator's configuration complexity (e.g., GPU or hardware core) and interface/interconnect option (PCIe, Infiniband, AMBA).

2) System Memory for User Data: When an application issues a *malloc()* system call, then a buffer object is created that is made available in its own virtual address space as a contiguous memory area. However, the allocated space may span physical memory pages that are not contiguous; this may occur even if the requested buffer size is smaller that system's page size, commonly set to 4KB. For instance, in the context of processing full high definition (FHD), or 1080p, which is 1920 pixels wide by 1080 pixels tall (2.1 megapixels), with 32 bits/pixel when loaded at memory 8,294,400 bytes are required, or 2025 pages of 4KB each. Even though it is preferable to use physically contiguous pages in memory both for cache related and memory access latency reasons this cannot be guaranteed when a user-level application makes *malloc()* calls.

The key idea is that in several application domains which can benefit through accelerating a computational intensive task, the memory space that is allocated at user space will be used throughout the offload process. Hence, the virtual to physical space translation that an IOMMU can perform during the acceleration process can be done even before the acceleration begins, since we know beforehand both the source and the result data buffers. Consequently, instead of using an IOMMU, the MMU of the CPU can perform the virtual to physical translation. Essentially, the physical address space of the non-contiguous space can be identified at kernel level and in addition, to allocate contiguous space larger than the page size is also possible at kernel space through using the *dma_alloc_coherent()* or *dma_alloc_noncoherent* calls.

To address offload data management for data that are allocated in system memory the GPPU infrastructure can employ the SysTem Memory, or *STM* method, which requires a Generic Address Translation Table, called *GATT* hereafter, as shown in figure 4. In combination with a kernel driver and AQLSM, the GATT can fetch from the system memory the translations of pinned pages and program the accelerators accordingly. The GATT can be programmed to operate in non-paged mode and in paged mode. The first mode is used when the data are stored in memory contiguously, therefore an accelerator's DMA has to be programmed only once. The second mode is used when the data span more than one page. In this case GATT is responsible to configure an accelerator's DMA upon interrupt on a per page basis, both for the incoming and for the processed data stream. Notice that in this second mode the pages translations are stored in a contiguous buffer in memory. In the scope of this STM method, the AQLSM

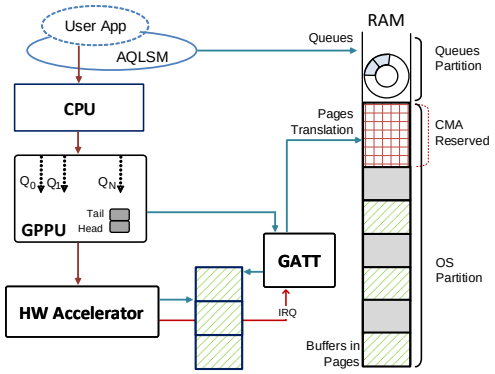

Fig. 4. Dispatching jobs to accelerators through the GPPU infrastructure; the kernel driver with the aid of the host MMU translates the virtual address space of the data buffers that are allocated at user space and stores them in the cross-shaded memory buffer, while the GATT mediates to provide the illusion of a contiguous space to the accelerator

runtime that runs at user space interacts with a kernel space driver only at the initialization of the offload process. The user application assigns to the AQLSM runtime to allocate data buffer and to discover the physical address of this buffer, which may be fragmented in many pages. This particular driver in addition to the virtual to physical address translation must handle cache effects, using standard Linux kernel facilities. If

Copyright (c) 2019 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCAD.2019.2907912

there is no hardware to provide cache coherency, the allocated memory space must be non-cacheable, or must be flushed or invalidated before giving control to a hardware accelerator. The AQLSM runtime fills in the command packet the physical address of the buffer with the translations and then triggers the GPPU with the doorbell signal. As shown in figure 4, the GPPU communicates both with the device to assign a job and with the GATT to configure it for the lifetime of this job.

IV. GPPU System Design

We designed the GPPU infrastructure on an 64-bit ARMv8 ARM Development Platform (ADP) named Juno-r1 [31], which is extended with the LogicTile 20MG FPGA board [32]. Table I summarizes the SoC features relevant to our design.

TABLE I
Overview of ARM Juno-r1 development platform

Dual cluster Cortex-A57	2MB L2 cache, NEON and FPU, underdrive: 600MHz, nominal: 900MHz, overdrive: 1.15GHz
Quad cluster Cortex-A53	1MB L2 cache, NEON and FPU, nominal: 650MHz
CCI-400 int. interconnect	cache-coherent, 128-bit, 533Mhz
NIC-400 ext. interconnect	64-bit, 400MHz, external AXI ports using Thin-Links cores

The LogicTile utilizes a Xilinx Virtex-7 FPGA, the XCV2000T device. The platform SoC LogicTile interconnect is realized through using the ARM TLX-400 Thin-Links AXI master and slave interfaces inside the LogicTile site [33]. At the default clock frequency of 61.5MHz, the operating bit rates are:

- Master interface (CPU-to-LogicTile): 68Mbps in the forward direction, 78Mbps in the reverse direction.
- Slave interface (LogicTile-to-CPU): 246Mbps in the forward direction, 305Mbps in the reverse direction.

The platform SoC runs the ARM Landing Team's Linux kernel v4.8 release. Next, we present the baseline design using the same hardware accelerators as in the GPPU system.

A. Baseline Accelerators

Figure 5 shows the FPGA baseline block design attaining the goal to realize two different types of hardware accelerators, which are integrated through a kernel-based driver dispatching. We developed one hardware accelerator for matrix multiplication and one for image edge detection using Vivado High-level Synthesis (HLS) tools [34]; to investigate the impact of dispatching to multi-threaded accelerators, we instantiated four image edge detection blocks. We consider each type of hardware accelerator as an acceleration group.

Table II summarizes their main characteristics. The hardware accelerators communicate via the ARM proprietary Thin-Links interface with the Juno SoC CCI-400 cache coherent interconnect [35]. The Juno r1 platform does not support an IOMMU for LogicTile-hosted components. Hence, the accelerators cannot transparently refer to system virtual address space and they can access only the physical address space.

The key concept in this baseline design is that the source data have to be copied from user to kernel space in order to be visible to the accelerators and for the same reason the resulting

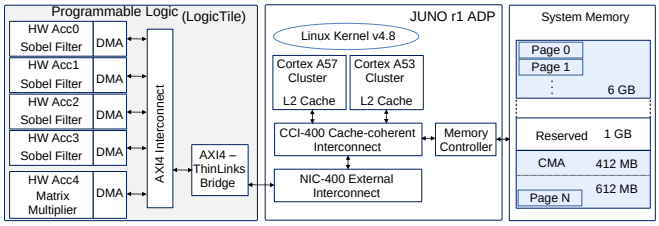

Fig. 5. Physical organization of the baseline architecture in ARM JUNO R1 equipped with the LogicTile 20MG FPGA

TABLE II
Hardware accelerators characteristics

Functionality	Characteristics
Matrix multiplication	Integer multiplication, max matrix size 100×100
Sobel edge detection	Convolutional image processor with kernels size 3×3, max image size (1920×1080)

data are copied from kernel to user space. The user space data are located in memory arranged in pages, which may be non-contiguous. Kernel space data are stored in Contiguous Memory Allocator's (named CMA) reserved partition, which allows for contiguous memory allocation. Data accesses are coherent between the CPUs and the programmable logic thanks to the CCI-400, which snoops the CPU caches.

The default kernel configuration provides only 4MB of contiguous memory space. The maximum contiguous space is calculated as $2^{MAX\_ZONE\_ORDER-1} * PAGE\_SIZE$, where the page size is 4KB. Thus, by changing the MAX_ZONE_ORDER setting in the kernel configuration file from 11 to 13, the kernel CMA can allocate up to 16MB of contiguous memory space; this is required to support image filtering of maximum size (i.e., FHD), one buffer for source and one for outcome data.

Baseline Dispatching Driver: The baseline driver communicates with the hardware cores to configure and offload tasks to them. The offloading is achieved in the following way.

- the driver allocates two contiguous data buffers in kernel space using the system's CMA, one buffer for the source data and one for the results
- the driver copies the caller's source data from user to kernel space via the *copy_from_user()* call
- then, programs the accelerator accordingly; notice that the driver is aware of the physical address of the buffer that is allocated in the first step
- the driver polls the accelerator to find out if the offloaded task is completed
- when the offloaded task has completed, the driver copies the result data from kernel to user space using the *copy_to_user()* call and frees the allocated buffers

The driver establishes synchronous communication when interfacing the hardware accelerators; when an application thread acquires an accelerator and offloads a job, blocks until job's completion. To manage the jobs per acceleration group, the driver uses logical buffers. The number of buffers is dynamically allocated and can be configured from a single buffer up to twice the number of accelerator instances inside a group; namely, the driver can assign up to eight buffers for the Sobel accelerators and up to two buffers for the matrix multiplication accelerator.

Copyright (c) 2019 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCAD.2019.2907912

B. GPPU Implementation

We integrated the GPPU with the hardware accelerators in the LogicTile FPGA to realize a proof of concept heterogeneous system with effective packet-based dispatching. Queue structures reside in system memory while the corresponding queue context is maintained inside the GPPU as a register file. The accelerators access the source and outcome data buffers in system memory using either of the two methods.

- SMP: access directly the reserved applications' data partition using physical addresses (through AQLSM runtime); data buffers are contiguous and cache coherent
- STM: access directly the scattered user pages managed by the OS with the aid of GATT; data buffers are non-contiguous and cache coherent

Notice that since CCI400 lacks support for cache coherent view to the LogicTile link, accelerators access data by using read transactions as *ReadOnce* and write transactions as *WriteUnique* for the Inner Shareable domain [31], so that data are not cached locally for future use.

Figure 6 shows the block-level organization of five hardware accelerators that a GPPU can assign jobs to them through a set of buffers, named *active jobs RAM*. This RAM is logically

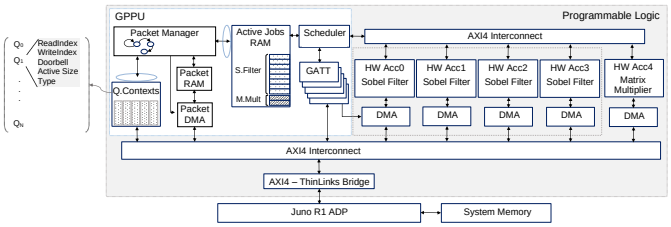

Fig. 6. GPPU architecture; the GPPU controls five hardware accelerators, four for image filtering and one for integer matrix multiplication.

divided into two partitions, one for the Sobel filter and one for the matrix multiplication acceleration group. We integrate eight job buffers for the Sobel filter and two job buffers for the matrix multiplication. Thus, in case that one accelerator is occupied by one active job, the GPPU can prepare the next ready job. A local scheduler abstracts the interfacing to the hardware accelerators and is in control of sending the jobs and of monitoring the acceleration process. The number of buffers to use in the active jobs RAM can be actually configured by the scheduler, as in the baseline dispatching method. In addition, in both SMP and STM modes the scheduler configures each dedicated GATT unit with the job context parameters; during the SMP mode the GATT operates in non-paged mode while during the STM mode GATT operates in paged mode.

The GPPU checks for each allocated queue if the doorbell is active and if the readIndex is different from the writeIndex. If a buffer is still available out of the buffers that are dedicated for the requested accelerator, then this job must be dispatched. Notice that the doorbell signal and the writeIndex are updated by the AQLSM runtime and are visible by the GPPU, while the readIndex is visible by the AQLSM and updated by the GPPU core, to avoid races. When the GPPU identifies that a job can be dispatched then the GPPU activates a private DMA engine, which can fetch all eligible packets from the queue. In the current realization the GPPU fetches a single packet (packets are retrieved one at a time) and dispatches it by extracting the fields from the packet that pertain to the particular accelerator. Fetching multiple packets via using DMA and processing them with a parallel processing GPPU gives an obvious advantage, and particularly in offloading of fine-grain jobs to high-performance or to multi-threaded accelerators. In favor of the latter case, another design option of the GPPU is to maintain the queues not in main memory, but locally inside the GPPU in a scratchpad memory; this option though, would impose size constraints on the queue and higher cost in terms of silicon area.

C. AQLSM Runtime

The AQLSM runtime exposes a simple API to the user and at the same time isolates different user applications and hides unnecessary complexity. The AQLSM runtime provides a GPPU discovery functionality to user applications, so that to expose the available accelerators in the system, named as *agents*. The AQLSM API represents agents using opaque handles. The application can traverse the list of agents that are available in the system using *aqlsm_iterate_agents()*, and query agent specific attributes using *aqlsm_agent_get_info()*. Examples of agent attributes include name, type of sub-nodes (CPU, HW accelerator, GPPU, Programmable accelerator), supported queue types, maximum number of queues and size.

In the current implementation the AQLSM runtime supports sixteen queues managed by a single GPPU; this GPPU represents an agent with five sub-nodes, four image filtering and one matrix multiplication accelerators acting as the accelerators. Inserting queues in the GPPU can reduce the amount of time the application spends stalling on the AQLSM API. The size of the queues is configurable and it has a direct impact on the parallelism that can be exploited by the GPPU. In general, larger queues enable more concurrency among the accelerators. In practice, contention for shared resources, and diversity in the accelerator execution time may impose unexpected limits to the benefits of parallelism.

One application should firstly initialize the AQLSM runtime via the *aqlsm_runtime_init()* function. During the initialization process AQLSM collects information, such as base physical addresses and size, about the available agents and the reserved AQLSM partition that is dedicated for the queues and data buffers. The Unified Extensible Firmware Interface (*UEFI*) gathers this information from the *device_tree* infrastructure and provides it to the Linux kernel. Figure 7 summarizes the STM-aware programming paradigm flow. The flow in the SMP method is free from step (4).

After the initialization process completes, the programmer allocates a queue via the AQLSM call *aqlsm_queue_create(queue_type)* which returns a valid *queue* object, if available, attached to *agent* (event (1)). Each queue is located in a system memory partition especially reserved for the AQLSM; the address of each queue's packet is calculated by the AQLSM with the following formula (*VA* stands for virtual address).

$$packetVA = queueVA + (packetIndex \ \% \ queue\_size)$$

Upon queue creation, the *queue* as a data structure will contain the following information. (i) a unique queue identification, namely the queue index, (ii) the base virtual address for the

Copyright (c) 2019 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCAD.2019.2907912

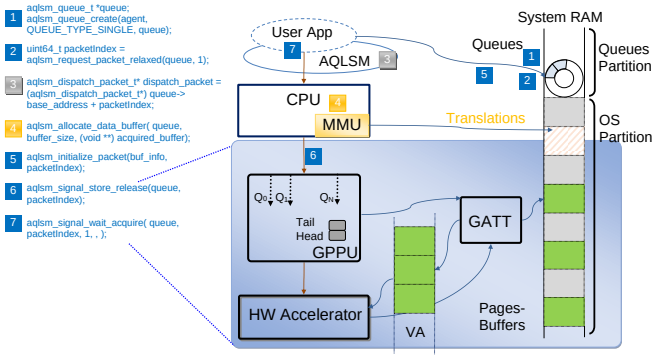

Fig. 7. Dispatching by using the AQLSM runtime with the STM method. The GPPU dispatches the packet and triggers the acceleration process between events 6 and 7. Event 4 invokes kernel-space to get the MMU service.

data buffer which corresponds to the specific queue (SMP mode), (iii) the size of the data buffer in bytes (SMP mode). The particular queue and context inside the GPPU are associated with the queue index, so that no false queue index can be intentionally or non-intentionally used. In the queue creation call, the queue packets are allocated and initialized as well.

Next, an available packet is requested via calling *aqlsm_request_packet_relaxed(queue, packets_amount)*, which returns the next packet number (*packetIndex*) in the queue (see Fig. 7 event 2), and internally AQLSM computes the address of the allocated packet (event 3). The application should allocate some space to be used as source and outcome data buffers. In the case of the SMP method, a partition in system memory is reserved for allocating physically contiguous data buffers and the available address space which corresponds to each queue (*queueAS*) is statically calculated by the AQLSM as: queueAS=(Total Available AS)/(Max Queues). If an application uses the STM method then the system (OS and AQLSM runtime) is responsible to maintain the data buffers, and the data allocation is done through the following function call (event 4 in Fig. 7).

```
aqlsm_allocate_data_buffer(int queueIndex,int
    buffer_size,void **acquired_buffer);
```

This function firstly pins the buffers to memory to avoid swapping and with the synergy of a kernel driver, stores the translations of the pages containing the buffer in a contiguous memory space. This call returns a data buffer identification (*buffer_id*). Notice that the user can issue only a single request to AQLSM to translate the virtual to physical addresses of a user space buffer via the processor's MMU and then re-use this allocated buffer throughout many offloading operations, to reduce the latency of the translation process. When the packet is launched, then the GPPU, with the aid of the GATT, will process the list of physical addresses properly, i.e., allow the accelerator to work in the virtual address of the application.

After filling the acquired buffers with data, the application requests the AQLSM to initialize the packet in memory via the *aqlsm_initialize_packet(aqlsm_packet_buffers_info_s buf_info, int packetIndex)* call (event 5). This *aqlsm_packet_buffers_info_s* structure is composed of the following fields:

```
struct aqlsm_packet_buffers_info_s {
int packet_type,
int accel_type,
int buffer_method,
int buffer_amount,
int *buffers_size,
int *buffers_type,
int *buffers_id };
```

where *packet_type* indicates the type of packet, either dispatch or barrier, and *accel_type* indicates the type of acceleration (possible types can be exported from *aqlsm_agent_get_info()*). The *buffer_method* includes information about the memory access method used to allocate the data buffers and might be one of the enumerations *AQLSM_BUF_SMP* or *AQLSM_BUF_STM*. The *buffer_amount* is the total number of allocated data buffers, *buffers_type* is a pointer describing whether the buffers contains source or outcome data, *buffers_size* is a pointer describing the size of data each buffer contains in bytes. The *buffers_id* is a pointer used only in the STM method, and contains the buffers id which the AQLSM uses to match the buffers with the appropriate translation list.

In summary, in order to launch a dispatch packet, the application must create a queue, request an available packet, allocate and prepare the data buffers, initialize the packet and then inform the GPPU that there is a new offload work available. The notification involves updating of the packet header as valid and signaling the doorbell to the GPPU (event 6). This is performed through the call *aqlsm_signal_store_release(queueIndex,packetIndex)*.

The AQLSM is then responsible to update the write index for this queue and the queue doorbell register in the GPPU. The execution of the launched job may start asynchronously while the application simultaneously submits additional packets in the same queue. If the application wants to get informed about the state of the launched packets, this can be achieved with the following function call (event 7).

```
aqlsm_signal_wait_acquire(int queueIndex,int *
    packetIndex,int packet_amount,int
    polling_interval_us,int max_time_to_wait_us);
```

The last parameter can be used to adjust maximum amount of time the function shall wait for the requested packets to complete; if zero value is passed, then the function blocks and returns only when the packets have completed.

AQLSM Drivers

The AQLSM runtime uses Linux user I/O (UIO) drivers to directly map the GPPU memory to a user space address range. Thus, user space applications have direct access to the GPPU memory and in particular to the GPPU contexts.This method eliminates the need for any mechanism to transfer packets back and forth between user space and kernel space. However, it is not possible to set up DMA operations from user space, or do contiguous memory allocation and direct cache control.

AQLSM runtime, in STM case, is complemented by a kernel space driver, which provides the translation (virtual to physical address) of user space allocated buffers, in order to give an accelerator the ability to seamlessly operate in application's virtual address space and essentially to access the corresponding system's main memory. The driver supports two commands, translate buffer pages and release buffer pages. When a translate buffer pages is issued, the driver translates and pins the buffer pages, generates an ordered list of their physical addresses (PAs) and stores them in a contiguous memory space using the services of the CMA and returns the

Copyright (c) 2019 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCAD.2019.2907912

PA of that list to the caller. When a release buffer pages is issued, the driver un-pins the translated buffers from memory and de-allocates the space where the list was previously stored. Finally, the driver has the ability to clear the system's cache if the system does not provide any cache coherence between the CPUs and external devices.

D. Synchronization

To achieve correct and efficient operation in a GPPU-enabled heterogeneous SoC, atomic operations need to be supported to guarantee safety when multiple initiators, hardware or software, concurrently access shared resources. Resolution of simultaneous accesses is required (i) among the GPPU hardware and the AQLSM Runtime, (ii) among applications that can share the same queue, i.e., access shared software resources through AQLSM calls and, (iii) and among the GPPU and the accelerators. AQLSM provides *full internal synchronization* in the sense that the caller user application does not need to perform any added synchronization operation.

Table III summarizes the race conditions that create the need to ensure atomic access since at least one initiator performs an update operation. For instance, a *data conflict* can occur when two user applications desire to allocate a new queue and read the queue status concurrently; the allocated queues are maintained as a bitmap vector inside a GPPU register. The two applications cause a *data race*, which can produce the allocation of the same queue unless the AQLSM handles this event using an atomic read-and-update to ensure that the entire operation sequence is indivisible. In total three potential cases can be raised, while two different locks are employed hereafter, since, even if cases *R1* and *R2* occur concurrently, they access the same location. Multiple applications residing in different clusters (A53 and A57) can simultaneously signal doorbells in the same GPPU, which is a race-free operation since the GPPU AXI4-Lite interface serializes the writes triggered by the AQLSM call of each application. Ordering semantics of doorbells are determined by the arbitration policy set in the CCI400 cluster interconnect and by the AXI4-ThinLinks bridge (see figure 6).

TABLE III
RACE CONDITIONS REQUIRING SYNCHRONIZATION OPERATIONS (VIA ATOMIC LOCKS IN CRITICAL SECTIONS)

Name	Concurrent operations	Comment
R1	Different apps request to allocate a new queue	Ensure race-free allocation of a queue
R2	App destroys a queue and app requests for a new queue	Avoid false allocation and false destruction
R3	Different apps request a write index in a shared queue	Ensure race-free allocation of a write index

We used the inherent support of ARMv8-specific instructions for exclusive memory access to implement atomic operations, using shared variables among multiple applications. The inline assembly code shown in figure 8 enables the lock and unlock of a mutex in user space. We selected the user space locks after we compared our implementation with two of the locking mechanisms that Linux kernel already provides. Figure 9 depictes the performance of different locking solutions on ARM big.LITTLE (big: 2 cores A-57, little: 4 cores A-53) architecture, where the big core operates in the minimum and

```
__asm( "ldaxr %x[old_val],[%x[ptr]]\n"
"cbnz  %x[old_val], 1f\n"
"stlxr %w[error], %x[lock], [%x[ptr]]\n"
"1:\n"
:[error]"=&r" (error)
:[old_val]"r" (old_val),[ptr]"r" (ptr),
 [lock]"r" (0x1)
:"cc", "memory" );
```

```
__asm( "ldaxr %x[old_val], [%x[ptr]]\n"
"cmp   %x[old_val], #1\n"
"bne   2f\n"
"stlxr %w[error], %x[unlock], [%x[ptr]]\n"
"2:\n"
:[error]"=&r" (error)
:[old_val]"r" (old_val),[ptr]"r" (ptr),
 [unlock]"r" (0x0)
:"cc", "memory" );
```

Fig. 8. User-space mutex lock (left); atomically load *ptr, if '0' atomically store '1' (a.k.a. lock), else exit. User-space mutex unlock (right); if '1' atomically store '0' (release lock), else exit.

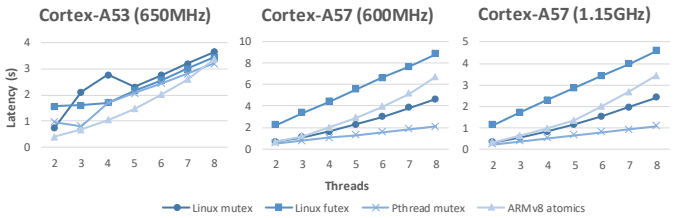

Fig. 9. Benchmarking lock mechanisms on Juno r1 SoC; average latency when scaling from two to eight threads with step one for one million atomic update operations each.

maximum frequencies. The user space locks using the ARMv8 Load-Acquire / Store-Release exclusive instructions show the best performance in the little cluster.

V. EVALUATION

We evaluate the proposed GPPU infrastructure (SMP and STM methods) against the baseline, using the implementation on Juno r1 development platform as described in section IV. Each dispatching method leverages the available accelerators by employing two different use cases, *Sobel Edge Detection*(*SED*) and *Matrix Multiplication*(*MM*). The objective is not to assess the effectiveness of the accelerators themselves with regard to the selected benchmarks, but to demonstrate the efficiency of user-level dispatching in synergy with hardware-assisted GPPU dispatching while offering an easy programming framework. By default, eight job buffers are enabled for the SED use case and two buffers for the MM.

The A57 dual-core cluster is configured in overdrive mode, i.e., 1.15GHz, during the evaluation. Considering the bandwidth constraints of the Thin-Links, in principle, we designed modest-throughput accelerators, instead of a single accelerator that could consume the full bandwidth of the Thin-Links. Thus, it makes sense to dispatch multiple jobs that can progress in parallel. Instead of integrating the high-end version of the SED core that we developed to achieve 305MB/s processing rates, we integrated a SED core version that can process an image at an average of 76 MB/s and a matrix multiplication core that delivers an average processing throughput of 6.8 MB/s, including the DMA latency.

For each dispatching method the user application consists of one master thread and a number of worker threads. In both SED and MM the master thread loads the data from a solid state disk to memory. Then, the worker threads do memory copies to initialize their private data pool; we call these operations as "data loading". In SMP and STM methods, through the AQLSM, a worker thread allocates its private queue and frees this queue after the dispatched job is complete.

Copyright (c) 2019 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCAD.2019.2907912

With the aid of the GPPU infrastructure, jobs are dispatched in a non-blocking fashion, while in the baseline method, job dispatching is non-blocking among threads and blocking for consecutive jobs within each thread, as far as there are available job buffers. Figure 9 depicts how job dispatching by two threads may incur latencies in time when blocking and non-blocking is employed on the A57 cluster. The shaded area marks the under-utilization of the physical accelerators when the blocking method is used.

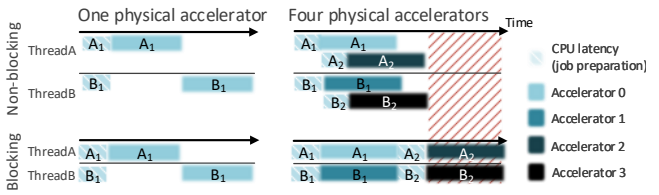

Fig. 9. Blocking and non-blocking job dispatching for two worker threads, when one (left) or four (right) accelerators are available.

A. Performance Results

Figure 10 compares the performance between the three dispatching methods for the SED use case. Both SMP and STM outperform the baseline for any number of jobs. This is due to distributed queue management and to hardware job scheduling, which enables non-blocking job dispatching and thus fully utilize the available acceleration system. On the other hand, the baseline method that uses blocking job dispatching in the case of a single worker thread, is not capable of hiding the initialization or completion delays in every case. As the number of jobs scales, the STM shows rapid improvement while later the gain becomes less impressive; this happens due to reusing the same job buffers and hence, the same page translations (virtual-to-physical) for offloads in batches of more than sixteen jobs. Notice that even in the case of a single thread, by dispatching many packets and by enabling multiple buffers in the Active Jobs RAM, the GPPU enables concurrent utilization of the Sobel accelerators.

Figure 11 shows the performance of the dispatching methods for the SED use case as we scale the number of accelerators from one to four.More precisely, we scale the number of buffers in the Active Jobs RAM and in the kernel driver (for the baseline case); each Sobel core processes the ready jobs in round-robin fashion. The number of worker threads is eight and each one dispatches 512 jobs. The measurements that are plotted refer to HD images (1280×720 pixels) on the left and FHD (1920×1080 pixels) on the right and the experiments are done on the A53 (top) and on the A57 cluster (bottom). As expected, when the number of buffers exceed the number of hardware accelerators, there is barely any performance gain for the SMP and STM methods; this holds for both the A53 and the A57 clusters. On the other hand, the baseline method shows performance gain even when we scale the number of job buffers from four to eight, since dispatching in the baseline method allows all the threads to initialize their jobs (by copying their data to the eight buffers). The STM and SMP methods give a 3.3× and 3.7× performance improvement over the baseline in the best case, in terms of jobs/sec. The worst case performance improvement is 1.5× and 1.7× respectively.

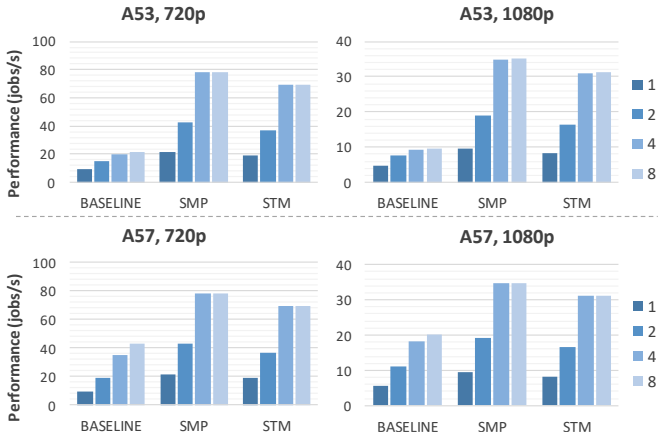

Fig. 11. Performance impact of scaling the number of job buffers when eight threads dispatch 512 SED jobs each.

With regard to the different CPU clusters, we receive exactly the same performance for the SMP and STM methods when we scale the number of job buffers from 1 to 8. This is due to the significantly lower CPU involvement compared to the amount of processing performed by the accelerators. On the contrary, regarding the baseline method, when we move the worker threads from the A53 to the A57 cluster, the

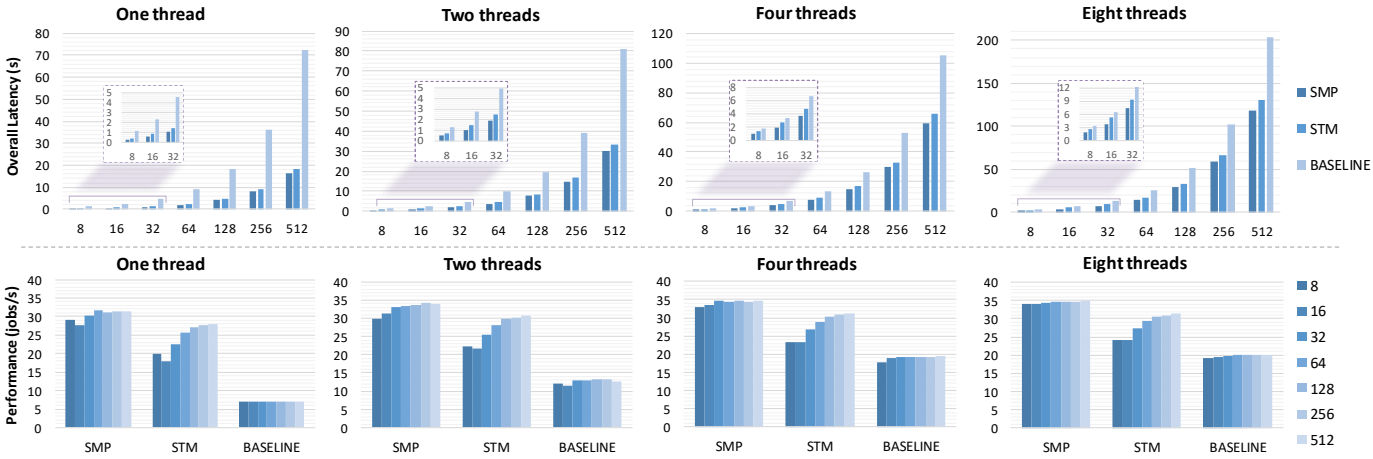

Fig. 10. Scaling the number of dispatched jobs from 8 to 512 per thread and the number of threads from 1 to 8 that execute on the A57 cluster; each job refers to SED processing of a FHD image

Copyright (c) 2019 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCAD.2019.2907912

performance improves from 1% to 105% for the HD images and 21% to 109% for the FHD images. Hence, SMP and STM methods barely depend on CPU's capacity in this scenario.

Figure 12 presents an analysis of execution latency when a single worker thread executes on the A57 cluster and dispatches 512 jobs to the Sobel accelerators in the SMP and STM methods. For fair comparison, to ensure non-blocking dispatching in the baseline method, we use four threads to dispatch 128 jobs each. Notice that breakdown delays

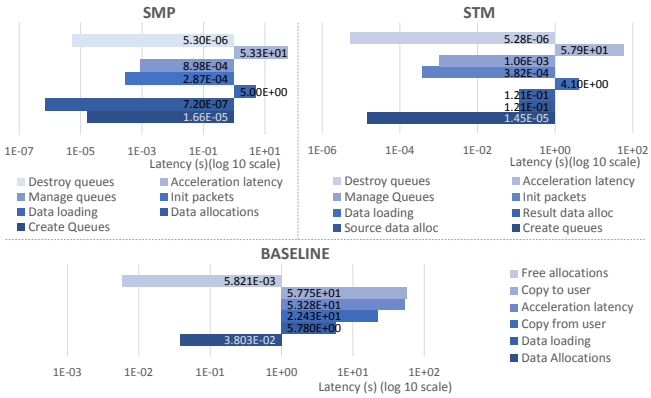

Fig. 12. Breakdown of the SED benchmark (on FHD images) executing on the A57 cluster and dispatching 512 jobs using all three dispatching methods; profiling is performed using hardware counters in the FPGA.

are cumulative over offloading 512 jobs and are shown in logarithmic (base 10) scale. In each plot the operations are listed in a bottom–up time sequence. The delay to initialize the values of each packet and the delay to manage the data buffers and queues are the most important in SMP. However, these latencies are negligible compared to other most dominant ones in the STM and baseline. In the STM method, the allocation of data, including pages translation and pinning, occurs in the initialization phase of offloading and has an important effect in the overall latency. Notice that since the size of each queue is sixteen packets, such latencies increase until the job number reaches sixteen; for batches larger than the queue size (i.e., sixteen), the translations are re-used since no more than sixteen different pending jobs can be maintained for a single queue. On the contrary, in the baseline case, the user-to-kernel and kernel-to-user space copying of data buffers incur dominant delays and scale exponentially with the workload size. As shown in the bottom plot, the transfer of the outcome data to user space costs 2.75× more time compared to user-to-kernel copies due to that the result data are transferred via *Write-Unique* type AXI4 snoop transactions from the accelerator, which causes the data to be stored in main memory.

As figure 12 shows, the actual acceleration latency in STM costs 8.63% more compared to SMP and baseline dispatch methods, and globally STM exhibits 6.76% higher latency compared to SMP. This extra latency is due to dynamic re-programming of the GATT that pertains only to the STM method. Finally, notice that the cost to load data differs in all three dispatching methods; this happens because during the acceleration processing that result data are stored in main memory, the single worker thread in both SMP and STM initializes jobs and does data loading at the same time. If we execute the same baseline scenario on the A53 quad core cluster, then the four threads perform data loading for the same number of jobs in just 3.87 sec.

Figure 13 shows comparative performance results based on the MM use case, in terms of jobs per second. As in the case of the SED benchmark, the SMP method outweighs the baseline method performance (jobs/s) in a range from 0.23% for two threads up to 65% for eight threads. For small amount

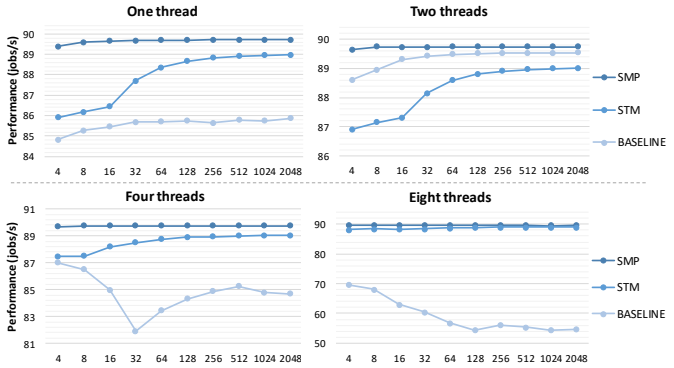

Fig. 13. Scaling the number of jobs offloaded to the single matrix multiplication accelerator and the number of threads running on the A57 cluster; a single job refers to integer multiplication of square 100×100 matrices

of workloads (one and two threads) the performance difference among the three methods is small. Actually in two threads, the baseline method gives 2.3% and 0.6% (for increased workload) better performance than the STM. For small matrices the kernel driver is faster for the copy-to-user-space procedure, compared to the virtual-to-physical translations in the STM method. The STM is more efficient by 0.5% to 63.6% for one, four and eight threads. Further, in the STM method, the "knee" that appears in all plots is due to the re-use of translated buffers for workloads larger than sixteen jobs.

B. Energy Results

To explore the impact of the GPPU in terms of energy consumption, we use actual sensors measurements, which we collect during our benchmarks. Juno board implements a set of platform energy meter registers in the IOFPGA [31]. These registers are updated every 100μs measuring instantaneous current consumption, instantaneous voltage supply, instantaneous power consumption, and cumulative energy consumption of the Cortex-A53 and Cortex-A57 clusters, Mali-T624 GPU cluster, and the fabric of the Juno r1 SoC. Monitor registers inside the IOFPGA are exposed through *sysfs* and accessed via the *hwmon* Linux kernel driver.

Figure 14 shows the energy consumption per job when scaling the number of image filtering offloads and concurrent threads. The cost of memory copies leads the baseline method to consume higher energy that ranges from 2.7× to 4.7×, compared to the proposed SMP and STM offloading strategies.

Figure 15 depicts the trade-off between performance and energy consumption when switching from A57 (big) to A53 (little) cluster. For all the dispatching methods we can see that there is a significant gain concerning the consumption of energy. Nevertheless, the impact in performance is insignificant for both SMP and STM methods, while for the baseline method performance degradation ranges from 41% to 124%.

Copyright (c) 2019 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCAD.2019.2907912

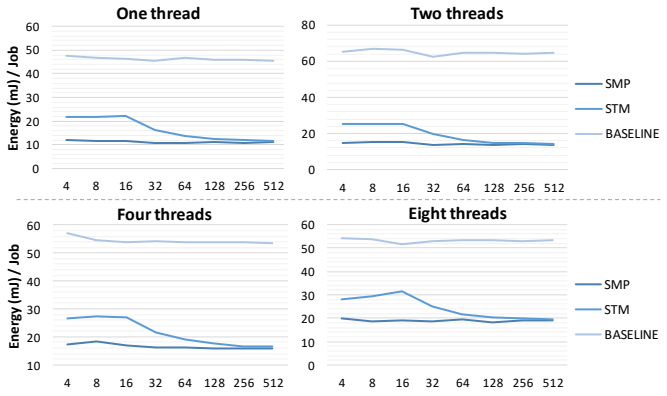

Fig. 14. Energy consumption per job for SED use case with SMP, STM and baseline methods when scaling the number of jobs from 4 to 512 and the number of worker threads from 1 to 8.

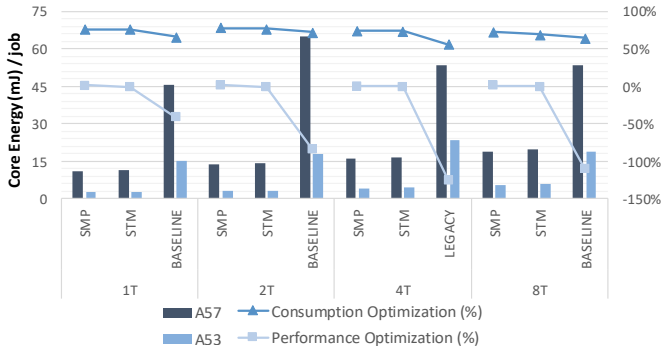

Fig. 15. Performance and energy consumption for SED use case for all three dispatching methods; number of threads scales from 1 to 8 for 512 jobs each.

A comparison for the SMP, STM and the baseline methods regarding the energy consumption is also conducted for the MM use case. By capturing the energy that IOFPGA reports for the same scenarios that were previously depicted in figure 13, the consumption gain for SMP is 5.1%-47.9% and for STM is 1.8%-50.7% compared to the baseline. In particular, in the case of two worker threads, where the baseline method proves to be slightly faster than STM (1.2% in average), the average gain in energy consumption for the STM over the baseline is 17.2%, ranging from 3.6% to 26.8%.

Figure 16 shows the trade-off between performance and energy consumption when switching from A57 to A53 cluster. As expected, when we switch the applications from A57 to A53, the energy consumption is optimized for all the methods. On the other hand, while the performance ranges almost at the same level in the SMP and STM cases, for the baseline method, the performance reduces by 2.3% (one thread) or improves by 5.4% (four threads) and 19.7% (eight threads); the latter results from the high intensity in communication with the kernel driver due to fine-grain jobs, and the fact that the A53 cluster offers two more physical processors.

In both use cases and for all methods we do not consider the energy required in the LogicTile, since the control operations of the GPPU are both infrequent and require data exchange with system memory in units of 64 bytes, which is negligible compared to the amount of processed data. To evaluate the energy overheads of the GPPU infrastructure from the FPGA prototype perspective, we used the Xilinx XPower Analyzer

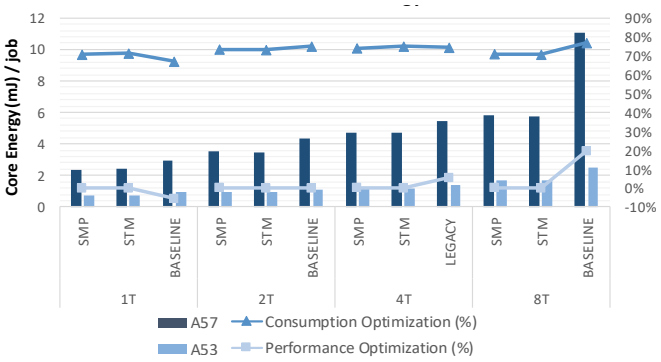

Fig. 16. Performance and energy consumption for MM use case for all three dispatching methods; number of threads scales from 1 to 8 for 2048 jobs each.

tool to extract the power figures of each hardware component. Through setting both toggle rate and static probability to a value of 50% we collected the dynamic power consumption that is shown in the second column of table IV.

TABLE IV
FPGA COMPONENTS LATENCY, POWER AND NORMALIZED ENERGY

Component	Power (watt)	Latency FHD image (μsec)	Latency 100x100 matrices (μsec)	Energy Ratio FHD Image E_{comp}/E_{acc}(%)	Energy Ratio 100x100 matrices E_{comp}/E_{acc}(%)
GPPU	0.61246	25	25	0.014440	0.143751
GATT	0.18820	2322	2	0.416469	0.003071
Image filter	1.00822	104074	N/A	100	N/A
Matrix multiplier	0.94412	N/A	11164	N/A	100

The time duration that each IP component is active, was reported by real hardware AXI counters/timers instantiated in the design. Hence, table IV summarizes the results of energy assessment for our implemented design. The overhead of the GPPU in terms of energy is negligible, only 0.14% for small jobs (i.e., small matrices) and the overhead of GATT, in the STM method, is only 0.41% for large jobs (i.e., FHD image processing). Notice that the power consumed in the FPGA I/O pins and the corresponding TLX interface is not included, since it is the same across all methods.

C. Device Utilization Results

Figure 17 shows the utilized area of the hardware components. The GPPU blocks (including GATTs) occupy 29%, the image filter accelerators occupy 49% and the matrix multiplication accelerator occupies 7% and operate in 100MHz.

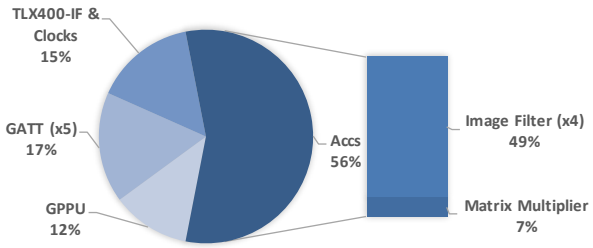

Fig. 17. Area cost (%) of each component and of the full acceleration system in the FPGA; Slice LUTs include both LUTs as logic and as memory, totaling 57906 LUTs. The GPPU includes also 16 BRAMs, the scheduler 16, the Image Filters 83 and the Matrix Multiplier 32 BRAMs.

D. Discussion

By using the GPPU framework, the distributed nature of queue operations, such as queue allocation, packet preparation and launching without OS system calls, together with

Copyright (c) 2019 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCAD.2019.2907912

hardware-assisted dispatching and batch-mode offloading enable shorter delays in job dispatching to hardware accelerators. While one dimension in achieving energy efficiency and optimal performance is through the optimization of special-purpose hardware accelerators, another important dimension is the efficient interfacing, in terms of software and hardware, of user applications to such systems. Through providing a unified virtual address space among the diverse computational units of a system, the programmer's productivity is greatly improved; all subtle interfacing, resource monitoring and scheduling to the accelerators is abstracted by the AQLSM. The benefit of GPPU to provide a unified packet-based API involves also limitations in terms of delay to de-capsulate packets and program the accelerator with the domain-specific parameters. As the level of heterogeneity between the hardware accelerators which are attached to a single GPPU increases, so is the complexity incurred for the GPPU. Further, the GPPU currently is not designed to decide which processes deserve access to the accelerator and for how long.

By scaling the number of queues as available resources for the applications, the GPPU can bring significant performance improvement only if more accelerators are integrated since the GPPU dispatch latency is orders of magnitude less compared to accelerator latency (see figure 12). The support of multiple independent queues incurs queue management and synchronization, which requires negligible complexity in both hardware and software components and most importantly incurs little perturbations among the different applications.

To maximize the system throughput co-located applications to shared accelerators with shared memory are examined in various works; by performing OpenCL kernel execution scheduling authors in [36] propose balancing of performance degredation. Recently, CuMAS [37] offers automatic overlapping of data transfers and kernel executions, but it focuses on scheduling multiple CUDA applications, rather than scheduling of a single application's data transfers. Scheduling is an additional direction by which the GPPU infrastructure can similarly be exploited to control system utilization or resource interference and prioritization. We intend to examine the applications' behavior and the impact of different GPPU scheduling algorithms to the system in future work.

SMP-based offloading delivers the best performance across scaling number of jobs, CPU type and accelerator type (single or multi-threaded, image filtering or matrix multiplication). This is due to the GPPU framework and mainly because the SMP strategy is free from any virtual-to-physical non-contiguous translation process. However, the downside is that we currently do not support memory management for the reserved data partition for such operation; a fixed-size memory region is statically assigned to each queue for the lifetime of the job. The STM strategy allows for full exploitation of system memory without the overheads of IOMMU address translation. The AQLSM runtime enables transparent adoption of either SMP or STM method and essentially removes the overhead of kernel calls since the only interaction with kernel-level driver occurs only in the initialization phase.

In the scope of GPGPU execution paradigm to take advantage of shared virtual memory (SVM) key feature across the CPU and the GPU, recent research proposed improvements inside page walk schedulers, necessary in reducing address translation overheads [38]. None of other schedulers i.e., sophisticated wavefront and memory controller schedulers attempt to tackle these overheads. The researchers apply batching of page table walk requests and larger IOMMU buffer size, which determines the size of the lookahead for the scheduler. For custom accelerators equipped with IOMMUs, recent works [39] propose to offload TLB misses to the page walker of the host core MMU, which effectively provides a unified virtual memory to accelerators but in a very intrusive way (requiring hardware modifications). Moreover, as SVM supports zero-copying which allows to pass only pointers between CPU and GPU for packet-based data access, it nevertheless requires using a separate memory allocator (e.g., clSVMAlloc()) instead of standard malloc() [40]. Additionally, frequent launch and teardown incurs overheads due to execution of heavyweight synchronization instructions to initialize the context registers at start and to make the results visible to the CPU side at teardown. To address the limitations of the zero-copy and memcpy approaches, more recent solutions introduce new memory hierarchy in multiGPU environments and extensions of the MOESI protocol [41]. As accelerators have now become first-class compute citizens, instead of employing such sophisticated techniques, unified memory addressing feature can provide great advantages with our proposed strategies which at the same time expose a much easier programming model.

VI. Conclusions

The key to exploiting accelerator-rich architectures is the efficient offloading of jobs; first, this involves architectural simplifications through providing support for devices' virtual address space without additional cost of IOMMUs and second, efficiency involves removing traditional OS system calls while at the same time providing a highly easy API and light runtime at the programmer side.

We have introduced the GPPU to support efficient communication between CPUs and accelerator components (programmable like GPUs, or custom hardware accelerators). The two GPPU core benefits are programmability simplification and communication latencies reduction. The developed GPPU is a hybrid component comprised of both mechanisms in hardware and in software, which facilitate efficient job dispatching. By exploiting user-level queuing, workload dispatching to hardware accelerators allows the removal of drawbacks related to copying objects through the operating system calls. We presented an optimized GPPU hardware that includes data structures supporting: (i) scaling number of queues, which are maintained in unified system memory space, (ii) consolidation of applications' scattered data that reside in non-contiguous memory space offering a contiguous device address space (thus eliminating the need of IOMMU for peripheral devices), (iii) synchronization mechanisms to achieve race-free sharing of multiple threaded applications that offload jobs to accelerators, and, (iv) hardware support for dispatching to hierarchical organization of accelerators. In addition, we developed the AQLSM Runtime which complements the GPPU

Copyright (c) 2019 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCAD.2019.2907912

innovative hardware and exposes an efficient programming layer to applications by reducing communication latency and programmability barrier. We believe that our GPPU infrastructure brings a homogenizing hardware layer to diverse accelerating components and a runtime to permit applications to be productively targeted to heterogeneous architectures while utilizing the available accelerators in an optimized way.

REFERENCES

[1] N. Brookwood, "AMD fusion family of APUs: Enabling a superior, immersive PC experience," 2010. [Online]. Available: www.amd.com/Documents/48423_fusion_whitepaper_WEB.pdf
[2] HSAFoundation, "HSA platform system architecture specification," revision 1.1, 21 Jan 2016.
[3] Y. Yang, P. Xiang, M. Mantor, and H. Zhou, "CPU-assisted GPGPU on fused CPU-GPU architectures," in *Proc. of the IEEE 18th Int'l Symp. on High-Perf. Comp. Arch.*, 2012, pp. 1–12.
[4] M. S. Orr *et al.*, "Fine-grain task aggregation and coordination on GPUs," in *41st Ann. Int'l Symp. on Comp. Arch.*, 2014, pp. 181–192.
[5] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu, "Performance gaps between openMP and openCL for multi-core CPUs," in *Proc. of the 41st Int'l Conf. on Par. Proc. Works.(ICPPW)*, 2012, pp. 116–125.
[6] T. Ramashekar and U. Bondhugula, "Automatic data allocation and buffer management for multi-GPU machines," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, pp. 60:1–60:26, Dec. 2013.
[7] T. B. Jablin *et al.*, "Dynamically managed data for CPU-GPU architectures," in *10th Int'l Symp. on Cod. Gen. & Opt.*, 2012, pp. 165–174.
[8] C. Augonnet *et al.*, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput.: Pract. Exper.*, vol. 23, no. 2, pp. 187–198, Feb. 2011.
[9] I. Gelado *et al.*, "An asymmetric distributed shared memory model for heterogeneous parallel systems," *SIGPLAN Not.*, vol. 45, no. 3, pp. 347–358, Mar. 2010.
[10] "GMAC-2: Easy and efficient programming for cuda-based systems," NVIDIA GPU Tech Conference GTC 2012, May 14-17, 2012.
[11] CUDA, "NVIDIA CUDA programming model."
[12] Y. S. Shao *et al.*, "Toward Cache-Friendly Hardware Accelerators," in *Proc. of the Sens. to Cloud Arch. Works. (SCAW)*, 2015.
[13] B. Wile, "Coherent accelerator processor interface (CAPI) for POWER8 systems, www-304.ibm.com/webapp/set2/sas/f/capi/home.html."
[14] Freescale, "MSBA8100 baseband accelerator, 2008."
[15] Analog, "ADSP-SC58x and ADSP-2158x series."
[16] Intel, "Enabling consistent platform-level services for tightly coupled accelerators." [Online]. Available: www.intel.com/content/dam/doc/white-paper/quickassist-technology-aal-white-paper.pdf
[17] Intel, "Intel quickassist technology, performance optimization guide," Num 330687, Rev 1.0, Sep. 2014.
[18] Y. Fujii *et al.*, "Data transfer matters for GPU computing," in *Proc. of the 2013 International Conference on Parallel and Distributed Systems (ICPADS '13)*, 2013, pp. 275–282.
[19] J. Cong *et al.*, "Architecture support for accelerator-rich cmps," in *Proc. of the 49th Ann. Des. Aut. Conf.*, 2012, pp. 843–849.
[20] F. Ji *et al.*, "DMA-assisted, intranode communication in GPU accelerated systems," in *Proc. of the 14th IEEE Int'l Conf. on High Perf. Comp. and Comm. (HPCC)*, 2012, pp. 461–468.
[21] D. Mbakoyiannis, O. Tomoutzoglou, and G. Kornaros, "Energy-performance considerations for data offloading to fpga-based accelerators over pcie," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 1, pp. 14:1–14:24, Mar. 2018.
[22] A. Kegel *et al.*, "IOMMU: Virtualizing IO through IO memory management unit (IOMMU)," ser. ASPLOS '16 Tutorials, 2016.
[23] G. Kornaros *et al.*, "I/O virtualization utilizing an efficient hardware system-level Memory Management Unit," in *Proc. of the 2014 Int'l Symp. on System-on-Chip (SoC)*, Oct 2014, pp. 1–4.
[24] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural support for address translation on GPUs: Designing memory management units for CPU/GPUs with unified address spaces," in *Proc. of the 19th Intl Conf. on Arch. Sup. for Prog. Lang. and Op. Sys.*, 2014, pp. 743–758.
[25] J. Power, M. D. Hill, and D. A. Wood, "Supporting x86-64 address translation for 100s of GPU lanes," in *IEEE 20th Int'l Symp. on High Perf. Comp. Arch.*, 2014, pp. 568–578.
[26] G. Kornaros and M. Coppola, "Enabling efficient job dispatching in accelerator-extended heterogeneous systems with unified address space," in *30th Int'l Symp.on Comp.Arch. & High Per.Comp.(SBAC-PAD)*, 2018.
[27] O. Tomoutzoglou, D. Bakoyannis, G. Komaros, and M. Coppola, "Efficient communication in heterogeneous SoCs with unified address space," in *11th Int'l Symp. on Rec. Com.-centric SoC*, Jun 2016, pp. 1–6.
[28] ARM, "ARM system memory management unit architecture specification," 2016, SMMU architecture version 2.0.
[29] Advanced Micro Devices, Inc., "AMD I/O virtualization technology (IOMMU) specification," 2011.
[30] J. Veselý *et al.*, "Generic system calls for GPUs," in *2018 ACM/IEEE 45th Ann. Int'l Symp. on Comp. Arch. (ISCA)*, 2018, pp. 843–856.
[31] ARM, "Juno ARM development platform SoC," 2015, Technical Reference Manual, Rev. r1p0.
[32] ARM, "ARM LogicTile Express 20MG," 2014, technical Reference Manual V2F-1XV7.
[33] ARM, "ARM Corelink TLX-400 network interconnect thin links," 2013.
[34] Xilinx Inc, "Vivado Design Suite User Guide High-Level Synthesis," Nov. 2015.
[35] ARM Corelink. CCI-400 Cache Coherent Interconnect Technical Reference Manual (ARM DDI 0470), 2013.
[36] S. Lee and C. Wu, "Performance characterization, prediction, and optimization for heterogeneous systems with multi-level memory interference," in *IEEE Intl Symp. on Workl. Char.(IISWC)*, 2017, pp. 43–53.
[37] M. E. Belviranli *et al.*, "Cumas: Data transfer aware multi-application scheduling for shared GPUs," in *Proc. of the Int'l Conf. on Superc.*, 2016, pp. 31:1–31:12.
[38] S. Shin *et al.*, "Scheduling page table walks for irregular gpu applications," in *45th Ann. Int'l Symp. on Comp. Arch.*, 2018, pp. 180–192.
[39] Y. Hao, Z. Fang, G. Reinman, and J. Cong, "Supporting address translation for accelerator-centric architectures," in *2017 IEEE Int'l Symp. on High Perf. Comp. Arch. (HPCA)*, Feb 2017, pp. 37–48.
[40] Y. Go *et al.*, "APUNet: Revitalizing GPU as packet processing accelerator," in *Proc. 14th USENIX Conf. on Net. Sys. Des. & Impl.*, 2017, pp. 83–96.
[41] A. K. Ziabari *et al.*, "UMH: A hardware-based unified memory hierarchy for systems with multiple discrete GPUs," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 4, pp. 35:1–35:25, Dec. 2016.

Othon Tomoutzoglou received both the MSc. and the BSc. degree from the Technological Educational Institute of Crete, Heraklion, Greece in 2014 and 2016 respectively. He is currenly a Design Engineer in research projects with the Technological Educational Institute of Crete. His current research interests include multicore and heterogeneous architectures, embedded and reconfigurable systems, RTL Design, high-level synthesis and operating systems.

Dimitris Mbakoyiannis received the BSc. degree from the Technological Educational Institute of Crete, Heraklion, Greece in 2014. He is currenly a Design Engineer in research projects with the Technological Educational Institute of Crete. His current research interests include heterogeneous architectures, embedded and reconfigurable systems, high-level synthesis and operating systems.

George Kornaros is an Assistant Professor of Informatics Engineering Dept. at the Technological Educational Institute of Crete, Greece, where he leads the Intelligent Systems and Computer Architecture Group. His research interests include multicore architectures, high speed communication architectures, and embedded and reconfigurable systems. Kornaros has designed single chip network processors for the industry, published more than 60 scientific articles, and edited the book "MultiCore Embedded Systems". He holds three patents and is a member of the Technical Chamber of Greece.

Marcello Coppola is the technical director at STMicroelectronics and has more than 20 years of industry experience focused on developing breakthrough technologies. He has a graduate degree in computer science from the University of Pisa, Italy. His research interests include HPC, IoT for education, cyberphysical systems, 5G, automotive technologies, and multi-core and many-core SoCs. Coppola has coauthored more than 50 scientific publications and held various roles in top international conferences and workshops. He holds 26 patents and is involved in multiple European research projects.

Copyright (c) 2019 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.